LiveCode Documentation Format Reference

Introduction

LiveCode uses a custom documentation format for the dictionary and for extension APIs. The format has very few special characters and is intended to maximise readability and modifiability.

Format

Documentation for a LiveCode API or documentation entry consists of a series of elements and their values, in the following form:

elementName : *content*

There are several standard types of *content*; which one may be used depends on the *elementName*.

Element names

Elements are optional unless otherwise specified.

Name (required)

The name of the API entry. The name must be a single word.

> *Note:* For inline docs, the name element is automatically generated

Synonyms

A comma-delimited list of synonyms for this API entry.

Type (required)

The type of this API entry. One of the following: (API-level)

- command
- function
- property
- message
- constant
- keyword
- control structure
- operator
- statement
- expression

(Glossary-level)

- library
- widget
- glossary
- object

The glossary-level entries have part of their content generated from the API-level entries.

> *Note:* For inline docs, the type element is automatically generated

Syntax (required for API-level entries)

A description of how to use or call this facet of the API. May just be the name itself, for keywords etc. An entry may have a number of Syntax elements, if there are variations within the same entry.

The following can be used to specify livecode syntax:

- `[optional]`
- `[repeated optional ...]`
- `{ variant 1 | variant 2 }`
- `<parameterName>`

For example, the syntax for the first variant of the `split` command:

```
split <variableToSplit> {by | using | with} <primaryDelimiter> [and <secondaryDelimiter
>]
```

describes the following possible usages:

- `split tVar by comma`
- `split tVar using return`
- `split tVar with "a"`
- `split tVar by comma and return`
- `split tVar using the rowdel and the columndel`
- `split tVar with tDelim1 and tDelim2`

The syntax for `answer file with type`:

```
answer file[s] <prompt> [with <defaultPath>] [with type <types> [or type <types> ...]]
 [titled <windowTitle>] [as sheet]
```

gives an example of the repetition notation, namely it allows something like

```
answer files with type tType1 or type tType2 or type tType3 or type tType4
```

> *Note:* For inline docs, the Syntax elements are automatically generated

All parameters described in the parameters section should appear in angle brackets in the syntax.

Summary (required)

A summary of the API entry being documented. The summary should be a single line. Anything more in-depth should be included in the description.

Associated

A comma-delimited list of names of other API or docs entries. Each one must be of type **object**, **glossary**, **library**, or **widget**. It is used to generate the docs for the entries to which it is associated.

In particular, every API entry in a library should have that library name in its Associated list.

Introduced

The LiveCode version in which the API entry was first available.

OS

A comma-delimited list specifying which operating systems the API entry is available for. One or more of the following:

- mac
- windows
- linux
- iOS
- android
- RPi
- html5

Platforms

A comma-delimited list specifying which platforms the API entry is available for. One or more of the following:

- desktop
- server
- web
- mobile

Example

Example elements should show how API entries are used in context. Each example will be displayed in the documentation viewer as a multiline code block. Where possible, an example should be entirely self contained, so that it is runnable as a block of code without any additions.

An API entry can have any number of Example elements.

Parameters

The parameters element itself does not do anything, but can be helpful for readability as a precursor to the parameter elements themselves. Parameter elements are specified in the following way: *paramName*(*paramType*) : *paramDescription*

The param type is optional, although may be helpful to include, and is required for further formatting of parameter descriptions to take place. The three types that will instigate further formatting are:

- enum
- set
- array

A parameter of **enum** type is one that expects one of a specified list of values. Similarly, a parameter of **set**type is one that expects a comma-delimited list of one or more of a specified list of values. After the param description of the **enum** or **set** type, these values should be listed in the following way

```
    - value1 : description of value1
    - value2 : description of value2
    - ...
```

etc.

After the description parameter of **array** type, the format of the array may optionally be specified in the following way:

```
{
    key1 (keyType) : description of key1
    value1 (valueType) : description of value1
    key2 (keyType) : description of key2
    value2 (valueType) : description of value2
    ...
}
```

The key type and value type are optional. If one of the values is a sub-array, and the **array** type is specified, then the array specification may be nested as follows:

```
{
    key1 (keyType) : description of key1
    value1 (array) : description of value1
    {
        key2 (keyType) : description of key2
        value2 (valueType) : description of value2
        ...
    }
    ...
}
```

The indentation is not necessary, but is helpful for readability.

Any parameter specified using `<parameterName>` in the Syntax element should be described in the parameters section.

Value and return parameters

For simple return parameters, i.e. the direct return value of a function, use the Returns element. When a value is both an input and output value, for example the value of a property, use the Value element. The Returns and Value elements are mutually exclusive.

Some functions and commands in LiveCode change the value of 'the result' and the 'it' variable. In these cases there are also special return parameter elements It and The result.

All of these elements can have a type specified in brackets before the colon, as for parameter elements. The same comments about formatting apply to return parameters.

Description (required)

A full description of the API entry. The description may have markdown elements which will be rendered in the documentation viewer.

References

A comma-delimited list of other API or docs entries that are referenced. The references element is essential for linking around the documentation viewer.

The references list should be in the form:

> *entryName* (*entryType*), *entryName* (*entryType*), *entryName* (*entryType*), ...

Tags

A comma-delimited list of tags for this entry. The documentation viewer can filter entries by tag.

Other documentation syntax

The Description element is the only one that allows standard markdown. There is only (essentially) one other syntactic form allowed in API documentation, which is enclosing text in angle brackets <>. There are three purposes of the angle brackets syntax:

- Specifying occurrences of the name of the current API entry.
- Specifying occurrences of the names of parameters.
- Specifying links to other API entries or documentation.

In the first two cases, applying the syntax merely consists in enclosing the relevant occurrences in angle brackets, eg to reference a parameter *pParam* in the entry named *thisEntry*,

> This is a sentence in which <pParam> ought to be marked as being a parameter in the API entry for <thisEntry>

To specify a link to another API or docs entry, the latter **must** be included in the References element. If there is only one item in the References element with a matching name, then it suffices to enclose the referencing text in angle brackets.

> This will link correctly to the <other> function's API entry provided the References element contains only one item with the name "other".

If the referencing text is different from the name of the entry, such as if it is pluralised or possessive, include the referenced entry name as a label inside the angle brackets.

> This will also link correctly to <other|other's> API entry, but display as "other's" in the documentation viewer.

If the target name is shared by two references, then the link should be disambiguated by including the type in brackets:

> We might want to link to the <other(function)> function and the <other(command)> command in the same entry, if the references contain "other (function)" and "other (command)".

It is possible to do both of the above at the same time:

> This sentence contains links to <other(function)|other's> and <other(command)|other's> dictionary entries.

Separate docs files

To specify the documentation for a library in one separate file, it must have the following structure:

Library: *libraryName*

Summary: *librarySummary*

Description: *libraryDescription*

Name: *entryName*

Type: *entryType*

Syntax: *entrySyntax*

Summary: *entrySummary*

...other elements associated with *entryName*...

Name: *anotherEntryName* etc.

Inline documentation

Documentation for LiveCode script and LiveCode extensions can be done in-line. In this case, the Name, Type, Syntax and Associated elements are auto-generated.

The code-block enclosed in /* / immediately preceding the handler or syntax definition is used for the other elements of the documentation for that entry. In LiveCode extensions, the types of parameters are also pulled from the associated handler.

Separate docs files examples

Dictionary example

Here is the dictionary document for the *textEncode* function:

```
Name: textEncode

Type: function

Syntax: textEncode(<stringToEncode>, <encoding>)

Summary: Converts from text to binary data.

Introduced: 7.0

OS: mac,windows,linux,ios,android

Platforms: desktop,server,web,mobile

Example:
textEncode("A","UTF16")

Example:
put textEncode(field "output","UTF-8") into tOutput
put tOutput into url ("file:output.txt")

Parameters:
stringToEncode (string): Any string, or expression that evaluates to a string.
encoding (enum): the encoding to be used
- "ASCII"
- "ISO-8859-1": Linux only
- "MacRoman": OS X only
- "Native": ISO-8859-1 on Minux, MacRoman on OS X, CP1252 on Windows
- "UTF-16"
- "UTF-16BE"
- "UTF-16LE"
- "UTF-32"
- "UTF-32BE"
- "UTF-32LE"
- "UTF-8"
- "CP1252"

Returns: Returns the <stringToEncode> as binary data.

Description:
Converts from text to binary data.

The <textEncode> function takes text, and returns it as binary data, encoded with the sp
ecified encoding.

It is highly recommended that any time you interface with things outside LiveCode (file
s, network sockets, processes, etc) that you explicitly textEncode any text you send out
side LiveCode and <textDecode> all text received into LiveCode. If this doesn't happen,
```

```
 a platform-dependent encoding will be used (which normally does not support Unicode tex
t).

It is not, in general, possible to reliably auto-detect text encodings so please check t
he documentation for the programme you are communicating with to find out what it expect
s. If in doubt, try UTF-8.

References: textDecode (function)
```

Library example

```
Module: com.livecode.sort

Type: library

Description: This library consists of the sorting operations provided by the standard li
brary of LiveCode Builder


Name: SortListDescendingText

Type: statement

Syntax: sort <Target> in descending [text] order


Summary: Sorts <Target> in descending text order.
Parameters:
Target (inout list): An expression that evaluates to a list of strings.
Example:
    variable tTestList as List
    put the empty list into tTestList

    push "abcd" onto tTestList
    push 1 onto tTestList
    push "xyz" onto tTestList
    push 2 onto tTestList

    sort tTestList in descending order -- tTestList is ["xyz", "abcd", 1, 2]


Description: Text sort is performed by comparing string elements on a codepoint by codep
oint basis. Non-string elements come after all string elements in the sort order. The so
rt is stable, so that non-string elements are not re-ordered relative to each other.


Tags: Sorting

Name: SortListAscendingText

Type: statement

Syntax: sort <Target> in ascending [text] order
```

Inline examples

In general, writing inline docs has fewer requirements since several of the elements are auto-generated.

LiveCode Builder syntax example

```
/**
This library consists of the operations on lists included in the standard library of Liv
eCode Builder.
*/

module com.livecode.list

/**
Summary:        Returns the first element of <Target>.
Target:         An expression which evaluates to a list.
output:            The first element of <Target>

Example:
    variable tVar as List
    put the empty list into tVar
    push "first element" onto tVar

    variable tResult as Boolean
    if the head of tVar is "first element" then
        put "success" into tResult
    end if

Description:
Returns the first element of the list <Target> without modifying <Target>.
\`\`\`    the head of tVar\`\`\`
is equivalent to using the <IndexedElementOfList> operator with index -1,
\`\`\`    tVar[1]\`\`\`

References: IndexedElementOfList(operator)

Tags: Lists
*/

syntax HeadOfList is prefix operator with precedence 1
    "the" "head" "of" <Target: Expression>
begin
    MCListEvalHeadOf(Target, output)
end syntax

end module
```

LiveCode Builder handler example

```
/**
Summary: Logs the result of a test to the <xResults> list

Parameters:
pModule: The name of the module this test comes from.
pTest: The name of the test.
pResult: The result of the test
xResults: The ongoing list of test results

Description:
Pushes either the string "SUCCESS : <pModule>_<pTest>" or the string "FAILURE : <pModule
>_<pTest>" onto the results list,
depending on the value of <pResult>

*/

public handler testLog(in pModule as String, in pTest as String, in pResult as Boolean,
 inout xResults as List)
    variable tStringResult as String
    if pResult then
        put "SUCCESS" into tStringResult
    else
        put "FAILURE" into tStringResult
    end if

    push tStringResult && ":" && pModule & "_" & pTest onto xResults
end handler
```

LiveCode script handler example

```
/**
Summary: Extracts the inline docs from a .lcb file

pFile: The path to the .lcb file to extract docs from

Returns (string): A string consisting of all the docs for the library, and the syntax an
d handlers present in the .lcb file

Description:
<revDocsGenerateDocsFileFromModularFile> is used when packaging a widget to create its A
PI documentation.
It generates the Library and Type elements from the declaration in the <pFile> (either w
idget or library), and extracts
the comment block that precedes any initial declaration for use as the library-level Des
cription element.
It then extracts the comment blocks that precede syntax and handler definitions in <pFil
e>, and generates the
Name, Type, Syntax, and Associated elements for each entry, as well as the parameter typ
es.

Tags: Package building
*/

function revDocsGenerateDocsFileFromModularFile pFile
...
end revDocsGenerateDocsFileFromModularFile
```